AD-A090 033   MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
XLMS: A LINGUISTIC MEMORY SYSTEM,(U)
SEP 80 L B HAWKINSON   N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TM-173   NL

1 OF 1
AD A
090033

END
DATE
FILMED
11-80
DTIC

15

72

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-173

XLMS: A LINGUISTIC MEMORY SYSTEM

Lowell B. Hawkinson

September 1980

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TM-173	2. GOVT ACCESSION NO. AD-A090 033	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 XLMS: A Linguistic Memory System		5. TYPE OF REPORT & PERIOD COVERED
	14	6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TM-173
7. AUTHOR(s) 10 Lowell B./Hawkinson	15	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT/Laboratory for Computer System 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/Department of Defense 1400 Wilson Boulevard Arlington, VA 22209	11	12. REPORT DATE Sep[redacted]80
		13. NUMBER OF PAGES 55 12 56
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

knowledge representation
semantic network
linguistic memory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

1 LMS ("Linguistic Memory System") is a knowledge representation formalism particularly designed for representing knowledge that can be straightforwardly expressed in natural language. Fundamentally, it is a semantic network formalism, a formalism for managing interconnected objects in a highly-organized, network-like memory. XLMS is a particular LISP-based implementation of LMS, intended primarily for experimental use.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4-76-14

XLMS: A Linguistic Memory System

Lowell B. Hawkinson

Abstract. LMS ("Linguistic Memory System") is a knowledge representation formalism particularly designed for representing knowledge that can be straightforwardly expressed in natural language. Fundamentally, it is a semantic network formalism, a formalism for managing interconnected objects in a highly-organized, network-like memory. XLMS is a particular LISP-based implementation of LMS, intended primarily for experimental use.

Key words: knowledge representation, semantic network, linguistic memory.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. This research was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-75-C-0661.

Acknowledgements. I want to thank all those who have used the various implementations of LMS that have existed over the past several years. The feedback received from that usage has contributed greatly to the design of XLMS as here documented, and has served to inspire a new variant of LMS called PREP, which is now undergoing development. Those whose contributions deserve particular mention include: William A. Martin, Peter Szolovits, Glenn Burke, Gretchen Brown, William J. Long, William Swartout, Ramesh Patil, Byron Davies, and John H. Thompson. Most of all, I wish to thank Prof. Martin for his continued support and encouragement of this work, and for his substantial contributions to its content. All of its deficiencies, I should note, are solely my own responsibility.

August 1980

CONTENTS

1. Introduction 1

2. LMS Notation 3

2.1 Double-Quoted Spellings 3
2.2 Labels 4
2.3 Parenthesis-Delimited Expressions 5
2.3.1 Primitive Tie Letters 6
2.4 Equations 6
2.5 Square-Bracket-Delimited Expressions 7
2.6 Complexes 7
2.6.1 Attachment Specifications 8
2.6.2 Anaphoric Abbreviations 10
2.7 Itemizations 11
2.8 Computational Expressions 12
2.9 Absorption Forms 13

3. Basic LMS Operators 14

3.1 Operators Pertaining to Concepts 15
3.1.1 The $\subset$ Relation and Operators that Use It 16
3.1.2 Operators that Look For Concepts 19
3.2 Operators Pertaining to Attachments 21
3.2.1 Operators that Get or Look For Attachments 22
3.2.2 Operators Pertaining to Value Zones 23
3.2.3 Operators Pertaining to Attachment Relations 24

4. Specialized LMS Operators 26

4.1 Operators Pertaining to Sequences and Seq. Fragments 26
4.2 Low-Level Operators Pertaining to Nodes 30
4.3 The Canonical Ordering of Nodes 33
4.4 Operators Pertaining to Atomic Symbols 36
4.5 Operators Pertaining to Numerically Labeled Concepts 37

Appendix I. BNF Summary of LMS Notation **40**

Appendix II. The MACLISP Implementation of XLMS **44**

Index **47**

1. Introduction

LMS ("Linguistic Memory System") is a knowledge representation formalism particularly designed for representing knowledge that can be straightforwardly expressed in natural language. LMS may be viewed as a notational formalism, but is, more fundamentally, a semantic network formalism, a formalism for managing interconnected *objects* in a highly-organized, network-like *memory*. *XLMS* is a particular LISP-based implementation of LMS, intended primarily for experimental use.

The predominant type of object in an LMS memory is the *concept*. A concept has three essential components, known as the *ilk*, *tie*, and *cue*, which are its *immediate constituents*. The ilk and tie of a concept must themselves be concepts, while the cue may be either a concept, a sequence, a sequence fragment, or an atomic symbol.

A *sequence* is an object in an LMS memory, the *elements* (and immediate constituents) of which may be concepts, sequences, sequence fragments, and/or atomic symbols, arranged in some particular order. A *sequence fragment* is like a sequence, but is notationally and operationally distinct; it exists primarily to provide a "taxonomic structure" above sequences. Both sequences and sequence fragments may have any number of elements. The null sequence is distinct from the null sequence fragment.

In XLMS, concepts, sequences, and sequence fragments are all implemented in terms of a more primitive type of object, the *node*. A node has two essential components, known as the *genus* and *specializer*, which are *its* immediate constituents. The genus of a node must itself be a node, while the specializer may be either a node or an atomic symbol.

The following two constraints apply to *all* concepts, sequences, sequence fragments, and nodes in a particular LMS memory on a particular occasion.

(1) No two objects of one of these types may have the same combination of immediate constituents in the same order. Therefore, the immediate constituents of such an object, appropriately arranged, may serve as a *proper name* for it: something that identifies the object uniquely within memory, and thus provides a basis for locating it and notating it.

(2) No such object, except for **summum-genus**, may be a constituent of itself. (The *constituents* of an object are its immediate constituents plus all constituents of its immediate constituents.) The concept **summum-genus** is its own ilk, tie, and cue.

In addition to their essential components, concepts, sequences, and sequence fragments, as well as other (incidental) nodes, may have any number of objects *attached* to them. Associated with each attachment is an *attachment relation*, which indicates how the attached object relates to what it is attached to. Attachment relations are objects of a special type.

In addition to the above-described kinds of objects, XLMS also manages atomic symbols, though only insofar as they are used for various special purposes within XLMS--see the section entitled "Operators Pertaining to Atomic Symbols".

An important aspect of XLMS is LMS notation, a notation for objects and attachments. Concepts, sequences, sequence fragments, and nodes all have at least one form of notation that makes use of the fact that their immediate constituents, appropriately arranged, properly name them. Also, any object in an LMS memory may have any number of labels assigned to it. A *label* is a syntactically simple proper name assigned to an object *solely* for purposes of notation. Sometimes, in processing LMS notation, it is necessary to assign a label provisionally to a *dummy node*, a special type of object that is *not* a node, but may later turn into one.

LMS was originally developed as a fundamental component of the OWL system. Discussions of the theoretical background of LMS, in the context of OWL, may be found in Hawkinson [4], Martin [7-9], and Szolovits et al [11]. LMS has been used in several "knowledge-based" application areas, including dialogue processing [1], computer program writing [6], and drug therapy recommendation [10].

2. LMS Notation

LMS Notation is a notation for objects and attachments. Every *expression* in LMS notation represents some object, though any given object (in a memory) may be represented by many different notational expressions.

When an LMS notational expression is *absorbed* by LMS, each object and attachment represented therein is created in memory as soon as it is *determinable* from its representation (which may be immediately, at some future time, or never), if it does not already then exist. A represented object or attachment is not yet determinable if its representation includes an as-yet-unassigned label, whether directly or via an "anaphoric abbreviation"; see below. We speak of absorbing rather than *reading* LMS notation, because LMS notation, unlike LISP notation, cannot always be absorbed when it is read. In particular, LMS notation cannot be absorbed at compile time in the MACLISP implementation of XLMS, because nodes cannot be created at that time.

LMS notation is unrelated to LISP notation, except in that each can appear (properly delimited) within the other. Whenever LMS notation appears within LISP notation, it must be delimited by square brackets. Conversely, LISP notation can appear within an LMS notational expression only as (or as part of) the body of a "computational expression" (see below).

This chapter presents the syntax of LMS notation somewhat informally. A more precise account is given in the appendix entitled "BNF Summary of LMS Notation".

2.1 Double-Quoted Spellings

All atomic symbols (except for LISP's **nil**--see below) have LMS notational representations as *double-quoted spellings*. Any "normally-spelled" atomic symbol except **nil** may be represented in LMS notation by its spelling enclosed in double quotation marks, e.g., "**fire**", "**M.I.T.**", "**two-o'clock**", "**52**". (A *normal spelling* is a non-null string of letters, digits, hyphens, periods, and apostrophes. Letters in normal spellings are not distinguished by case, font family, face, or size; thus, for example, "**fire**", "**Fire**", and "**FIRE**" all represent the same normally-spelled atomic symbol. In XLMS, letters in normal spellings are represented by ASCII codes for upper-case letters, and also by atomic symbols corresponding to upper-case letters.) Also, any atomic symbol, whatever its spelling, may be represented in LMS notation by: *two* double quotation marks, followed by the spelling with any double quotation marks therein doubled, and terminated by a double quotation mark. Examples of this second format are ""**Thank you.**", ""**a+b**", and """", the last representing the atomic symbol whose spelling consists of a *single* double quotation mark.

Note that in addition to its double-quoted spelling representation(s), all atomic symbols except LMS's "**nil**" have *LISP* notational representations, e.g., **fire** or **|Thank you|**, that do not involve square brackets. (LMS's "**nil**" is distinct from LISP's **nil** to avoid not-found/**nil**-found ambiguities. In other words, LISP's **nil** is not used for "**nil**" so that **nil** can be used as an unambiguous not-found value by an operator that might actually find and return "**nil**".)

2.2 Labels

Labels in LMS are similar in purpose and use to labels in an "assembly language". A label is a name assigned to an object *solely* for the purpose of representing it in LMS notation. *Global* (globally scoped) labels must be normally spelled, whereas *local* labels (not yet supported in any implementation of XLMS) must be normally spelled except for a prefix **$**. An object may have any number of labels assigned to it, though most objects will have no label at all.

If a label, say **foo**, has been or will be assigned to an object x, then **foo** (or **Foo** or **FOO**, since letters in label spellings are not case-distinguished) may be used as an LMS notational expression for x. A label may get *assigned* to a particular object as an immediate or eventual consequence of the absorption of an equation that defines that label, as discussed subsequently in the section on equations. Although a label may be assigned to any object, it is expected that most labels will be assigned to nodes; in fact, when XLMS must *commit* itself as to what type of object an as-yet-undefined label represents, it will always expect a node of some kind to be the eventual assignment. Note that it is very common for a concept with an atomic symbol cue (or for a node with an atomic symbol specializer) to have a label of the same spelling as its cue (specializer).

A *numeric label* (a label that is *spelled* like a number, e.g., **17**, **-235.6**, and **1.23E-5**) is distinctive as far as assignment is concerned in that it is intrinsically assigned to some particular "numerically labeled concept", whose immediate constituents are determined by XLMS.[1] A numeric

[1] Incidentally, note that a given number might be represented by many distinct concepts, to say nothing of other objects, within an LMS memory.

label need never be explicitly assigned, but when it is explicitly assigned, it must be assigned to its implicit assignment.

2.3 Parenthesis-Delimited Expressions

Any node whatever[2] may be represented in LMS notation by a *parenthesized pair*, an expression of the form (*genus specializer*), where *genus* and *specializer* are LMS notational expressions for the the node's genus and specializer, respectively. For instance, if the genus and specializer of x are labeled **foo** and **bar**, then **(foo bar)** may be used to represent x in LMS notation. Note that *genus* and *specializer* are usually, often necessarily, separated by a space.

A concept may be represented in LMS notation by an expression of the form (*ilk***tie cue*) or (*ilk* **tie cue*),[3] where

ilk is a label, an anaphoric abbreviation, or a delimited LMS notational expression[4] for the concept's ilk;

tie is a letter, a label, or a delimited LMS notational expression for the concept's tie; and

cue is any LMS notational expression for the concept's cue.[5]

For example, **(dog*s bull)**[6] is an LMS notational expression for a concept that might well represent the species of dog referred to in English by "bull dog".

[2] Note, however, that representing a *concept* node in this way is *not* sufficient to indicate that it is a concept. See a later footnote in this section.

[3] A space (or equivalent--see below) is allowed before the asterisk so that a line break may occur there. This is especially useful when *ilk* is a large expression.

[4] A *delimited* LMS notational expression is one that is delimited by matching parentheses or square brackets.

[5] The concept (*ilk***tie cue*), where *tie* is *not* a letter, is implemented in XLMS as the node ((*ilk tie*) *cue*), marked so as to distinguish it as a concept node. Where *tie* is a letter, (*ilk***tie cue*) is implemented as the marked node ((*ilk* [**tie*]) *cue*). No assurance can be given, however, that ((*ilk tie*) *cue*) will not at some point be changed, even in XLMS, to, say, (*ilk* (*tie cue*)).

[6] Or (**dog**$_s$ **bull**), since in formal publications, (*ilk***tie cue*) could also be written (*ilk*$_{tie}$ *cue*).

2.3.1 Primitive Tie Letters

When *tie* is a letter in an expression of the form (*ilk***tie cue*) or (*ilk* **tie cue*), it designates a concept that serves as a primitive tie. Primitive tie letters are not case-distinguished; thus *s and *S both designate the same primitive tie. The primitive ties that are predefined in XLMS are as follows.

***s**	species
***t**	stereotype
***i**	individual
***r**	role-in
***u**	unique-role-in
***a**	appositive
***f**	function
***p**	partitive

Additional primitive ties may be defined; see the description of **define-primitive-tie**.

A primitive tie may be represented *per se*, in either LISP or LMS notation, as [**letter*]. Thus, for instance, [*r] is both a LISP and an LMS notational expression for the role-in primitive tie.

2.4 Equations

An *equation* is of the form $a = b$, where a and b are LMS notational expressions. An equation asserts that each of its sides represents the same object, i.e., it is an **eq** assertion.[7] An equation may be used as an LMS notational expression for the object (in memory) represented by each of its sides, as is the case, for example, in (**president*u U.S. = (country*i "U.S.)**).

When an equation with a label on either side is absorbed, the label is thereby *defined*, which means that it will be assigned to the object represented by the other side of the equation as soon as that object is determinable, if ever. (Recall that a node represented in terms of labels is determinable only after all those labels have been assigned.) Where *both* sides are labels, this rule applies to each of the labels individually. If an as-yet-undefined label has been absorbed as an LMS notational expression, or if the ultimate assignment of an already-defined label is not yet determinable, the label will have been provisionally assigned to a

[7] **eq** is the primitive LISP predicate that tests whether its two operands both represent the same object in memory.

dummy node. A dummy node is *not* a node, but may, under certain circumstances, *turn into* a node that will be some label's ultimate assignment.

When the left side of an equation is a global label and the right side is a parenthesis-delimited expression in which the specializer or cue subexpression is to represent the atomic symbol of the same spelling as the global label, then that specializer or cue subexpression can be written as " (pronounced "*ditto*"). Thus, for example, **apple = (fruit*s "apple")** may be abbreviated to **apple = (fruit*s ")**.

2.5 Square-Bracket-Delimited Expressions

Any object may be represented within LISP *or* LMS notation by enclosing an LMS notational expression for it in square brackets. A *square-bracket-delimited expression* appearing *in LISP notation* typically represents an object that can be directly (or LISP-dialect-independently) represented *only* in LMS notation: a concept, a primitive tie, a sequence, an attachment relation, the LMS atomic symbol "**nil**", an object represented as an absorb-time computation, an atomic symbol with one or more unusual characters in its spelling, etc. On the other hand, square-bracket-delimited expressions appearing *in LMS notation* are typically bracketed for the same sorts of reasons that lead one to parenthesize mathematical subexpressions: for clarity; for marking subexpressions that would not (or might not) be read as subexpressions because of precedence rules; and for overcoming specific syntactic restrictions, e.g., that which disallows an (unbracketed) equation as *ilk* or *tie* in an expression of the form (*ilk***tie cue*).

Note that nodes, like atomic numbers, "evaluate to themselves" in LISP and thus need never be quoted in LISP code. Hence, **[144]**, **[(dog*s bull)]**, and **[*s]** are well-formed LISP forms.

2.6 Complexes

An extension of the square-bracket-delimited form for representing objects permits the representation of attachments. (It also allows, somewhat incidentally, an arbitrary number of not-necessarily-related objects to be separately represented within a single LMS notational expression.) For example,

[blue c#e (color*r sky)]

asserts that **(color*r sky)** is to be a **#e** attachment on **blue**, and that **blue** is to be a **#c** attachment on **(color*r sky)**, where **#c** and **#e** are "attachment relation names"--see below.

An expression of the form[8]

```
[subject a01 a02 ...
        attachment-specification1 a11 a12 ...
        attachment-specification2 a21 a22 ...
        ... ]
```

is called a *complex*, where *subject* and the a_{ij} (of which there must be at least one) are LMS notational expressions, and where each *attachment-specification$_i$* (if any) specifies one-way attachment, two-way attachment, or non-attachment, as described below. Whether it appears in LMS or LISP notation, a complex represents what *subject* represents--some object *s*, say--and also asserts, for each a_{ij} for which *attachment-specification$_i$* exists and is other than just **#**, that the object represented by a_{ij} is to be an attachment on *s and/or vice versa*; see below for further details. Note that *all* the a_{ij} are absorbed when the complex is absorbed, even those (if any) for which either $i = 0$ or *attachment-specification$_i$* = **#**.

2.6.1 Attachment Specifications

An *attachment specification* is either an attachment relation name, a reverse-attachment specification, a two-way attachment specification, or **#**. An *attachment relation name* consists of a **#** prefixing either a single letter or a normal spelling of three or more characters, e.g. **#c** or **#characterization**. A *reverse-attachment specification* is like an attachment relation name except that the **#** appears as a suffix, e.g. **e#** or **exemplar#**. A *two-way attachment specification* consists of either (1) a reverse-attachment specification overlapping an attachment relation name in such a way that they share a single **#**, e.g. **e#c** or **exemplar#characterization**, or (2) a **#** prefixing two letters, each of which is the letter in some single-letter attachment relation name, e.g. **#ce** (which is equivalent to **e#c**). Note that letters in attachment specifications are not case-distinguished.

[8] The way in which an expression of this type is broken up into lines is irrelevant to its interpretation. Like all LISP and LMS notational expressions, the complex may have any combination of spaces, tabs, line breaks, "semi-colon comments" (in MACLISP), etc. inserted into it *wherever a space is allowed or required*. A space is allowed, and often required, between adjacent elements of a complex. See the "BNF Summary of LMS Notation" appendix for complete details.

An attachment relation name used as (or as part of) an *attachment-specification*$_i$ in a complex asserts that the objects represented by the a_{ij} that follow it are to be attached to what *subject* represents per the named relation. Conversely, a reverse-attachment specification used as (or as part of) an *attachment-specification*$_i$ asserts that what *subject* represents is to be attached to each of the objects represented by the a_{ij} per the corresponding relation. Thus, for example,

[dog #c barker carnivore e# barker carnivore]

asserts that the objects labeled **barker** and **carnivore** are to be attached as characterizations to the object labeled **dog**, and also that the object labeled **dog** is to attached as an exemplar to the objects labeled **barker** and **carnivore**. Note further that either

[dog exemplar#characterization barker carnivore]

or

[dog e#c barker carnivore]

would have the same effect on memory (after absorption) as the preceding notational expression.

The attachment relations that are predefined in XLMS and are not private[9] all have both single-letter and spelled-out names. They are as follows.

#c	**#characterization**
#e	**#exemplar**
#m	**#metacharacterization**
#x	**#inverse-metacharacterization**
#q	**#equal**
#v	**#value**
#r	**#role-in**
#h	**#has-role**
#f	**#function**
#a	**#applicable-function**
#p	**#predicate-type**
#k	**#Katz-feature**
#s	**#syntactic-name**
#n	**#semantic-name**

[9] The attachment relations used *privately* by LMS will not be discussed here, except incidentally, since anything private is subject to change without notice.

Additional attachment relations may be defined; see the description of **declare-attachment-relation**.

An attachment relation may be represented in either LISP or LMS notation by enclosing an attachment relation name for it in square brackets. Thus, both **[#c]** and **[#characterization]** are LISP *and* LMS notational expressions for the **#characterization** attachment relation. Note that attachment relations evaluate to themselves in LISP and thus need never be quoted in LISP code.

2.6.2 Anaphoric Abbreviations

Within the a_{ij} in a complex c, the subject of c may be represented *anaphorically*[10] by some number of consecutive uparrows or colons. An *uparrow anaphor* consisting of n consecutive uparrows represents the subject of the complex n levels up in the nest of square-bracketed expressions in which it (the uparrow anaphor) appears. For example, in the LISP form

```
(absorb
    [automobile
        [(trunk*u ^)
            [(interior*u ^)
                [(color*u ^) #c (color*u ^^^)]]]]),
```

the most deeply embedded square-bracketed expression is equivalent to **[(color*u (interior*u (trunk*u automobile))) #c (color*u automobile)]**.

The number of colons in a *colon anaphor* reflects how many levels of square bracketing the subject being referred to is removed from the outermost LISP-notation context. Thus a complex that is *not* itself embedded in a square-bracketed expression may have its subject represented as a single colon (:); a complex that is inside one set of square brackets may have its subject represented as a double colon (::); a complex that is inside exactly two pairs of square brackets may have its subject represented as a triple colon (:::); etc. For example, in the LISP form

[10] An *anaphor* is a syntactically simple reference to something expressed or implied earlier, or higher up, in some structure. Pronouns, for example, are anaphors.

```
(absorb
     [automobile
          [(trunk*u :)
               [(interior*u ::)
                    [(color*u :::) #c (color*u :)]]]]),
```

: represents what **automobile** represents, :: represents what **(trunk*u :)** represents, and ::: represents what **(interior*u ::)** represents.

Special abbreviations exist for certain LMS notational expressions involving anaphors. An LMS notational expression of the form (*b****r** *a*), where *a* is an anaphor and *b* a label or delimited LMS notational expression, may alternatively be written *ba*. Similarly, an LMS notational expression of the form (*b****u** *a*), where *a* and *b* are as before, may alternatively be written *ba***u**. Finally, an expression of the form ((*b****r** *a*)***l** *c*), where *a* and *b* are as before and where *c* is either a label other than **u**, a delimited LMS notational expression, or a double-quoted spelling, may be written *bac*. For example, **((wheel*r ::)*l left-front)** could alternatively be written **wheel::left-front**, and **[(color*u ^) #c (color*u ^^)]** could be written **[color^u #c color^^u]**.

2.7 Itemizations

A sequence of more than one element may be represented in LMS notation as an *itemization*

$$a_1, a_2, \ldots, a_n$$

where the a_i are LMS notational expressions for the elements, in order.

A sequence of one element may be represented by a *trivial* itemization [*a*,], where *a* is an LMS notational expression for the element. (Square brackets are not really part of a trivial itemization, but they are almost always used, for clarity if not out of necessity.) The null sequence has no representation as an itemization, but may be represented either by [] or by the label **null-sequence**.

A sequence whose elements are nodes in canonical order[11] may be represented in LMS notation as an *order-insensitive* itemization a_1 **&** a_2 **&** ... **&** a_n, where the a_i are LMS notational expressions for the elements *in any order*. An itemized set or "multiset" might well be represented canonically in LMS in terms of a canonically ordered sequence of its elements. By using an order-insensitive itemization, such a canonical set

[11] See the section, in a later chapter, on the canonical ordering of nodes.

or multiset representation can be notationally expressed in terms of its elements *with no dependence on which permutation of its elements is canonically ordered.* For instance, (**feature-set*1 heavy & rough**) and (**feature-set*1 rough & heavy**) are equivalent, whether **heavy** canonically precedes **rough** or vice versa.

A sequence fragment of one or more elements may be represented in LMS notation as an itemization

$$a_1, a_2, ..., a_n, ...$$

where the a_i are LMS notational expressions for the elements, in order, and where the terminating ... is part of the actual LMS notation. For example, **2, 3, 5, ...** represents the sequence fragment whose elements are **2, 3,** and **5**, in that order. The null sequence fragment may be represented by the label **...**.

In an itemization, none of the a_i may be an equation, because the comma or ampersand of an itemization has higher precedence than the equal sign of an equation. Also, an ampersand has higher precedence than a comma. Thus, for example, **[a = b, c & d]** would be interpreted as **[a = [b, [c & d]]]**.

2.8 Computational Expressions

Any object *x* may be indirectly represented within LMS notation (or, after adding square brackets, within LISP notation) as a *computational expression*: an LMS notational expression of the form !*e*, where *e* is a LISP form in LISP notation that evaluates to *x* at *absorb-time* (normally load-time).

Atomic numbers are readily representable in LMS notation as computational expressions, e.g., **!29** and **!4.71E6**.[12] LISP's **nil** may be represented in LMS notation by **!nil**. Computational expressions also provide yet other LMS notational expressions for atomic symbols; for instance, **!'fire** is equivalent, in LMS notation, to **"fire"**.

One disadvantage of representing objects in terms of computations is that such computations and/or their representations may be LISP-implementation-dependent.

[12] Note, incidentally, that numbers in LISP notation in the source text for LMS modules are interpreted in base ten by default.

2.9 Absorption Forms

An expression of the form [% a_1 a_2 ...] appearing in LISP notation is called an *absorption form*, and represents not a data object, but rather a LISP form which, when evaluated, will cause [a_1 a_2 ...] to be absorbed. The % in the absorption form (which must immediately follow the opening square bracket) effectively postpones absorb-time until evaluation time, thus making possible multiple absorption of the LMS notation within the form.

An absorption form that will (or may be) multiply evaluated typically contains one or more free variables[13] that might have different values on different evaluations of the form. Thus, for example, evaluation of

```
(loop for x in '(duck gull swan) do [% (bird*s !x) #c water-bird])
```

has the same effect as the absorption of

```
[[(bird*s "duck") #c water-bird]
 [(bird*s "gull") #c water-bird]
 [(bird*s "swan") #c water-bird]].
```

An absorption form may be used as a notation-based constructor. Instead of using explicit LISP code to construct one or more nodes (possibly with attachments), one can often use an equivalent (and almost always more readable) absorption form.[14]

[13] In the MACLISP implementation of XLMS, such free variables must, at present, be special variables.

[14] In the MACLISP implementation of XLMS, there is, at present, no special provision for efficient compilation of absorption forms. Thus absorption forms are substantially slower to evaluate than equivalent explicit LISP code.

3. Basic LMS Operators

The operators described in this chapter are sufficient for working experimentally with objects in an LMS memory. For non-experimental work, more specialized operators might be required, especially to achieve a high level of operating efficiency.

An operator is described by first displaying a *prototype application* for it, then discussing aspects of its use and behavior, especially those which are not inferrable from its name and from the dummy names for operands given in the prototype application in lieu of arguments. The name of an operator is chosen to suggest both what it does and what it may return as a value. Dummy names for operands are chosen so as to convey, as well as possible, assumptions about those operands; an operand named *concept*, for instance, may be assumed to be a concept.[15] Except as noted, arguments in applications of LMS operators must be LISP forms.

Also appearing in this chapter are descriptions of particular *iteration-driving clauses* that may be used in **loop** forms (see TM-169, "LOOP Iteration Macro" [2]). Such descriptions begin with a prototype iteration-driving clause of the form

(... **for** *variable* ...),

where a particular prototype *path clause* appears after *variable*. An actual **loop** form containing an iteration-driving clause begins with **loop** (usually), has a LISP variable name in place of *variable*, has one or more actual argument forms appearing within the path clause, and has some number of other clauses before and/or after the iteration-driving clause. Evaluation of such a form entails iterated evaluation of its body, with the specified variable bound, prior to each iteration, to the next value "along the specified path". Iteration may terminate for any of a number of reasons, one being that "the end of a path has been reached" (which may even precede the first iteration). Note that path values may be *listed*, in order, by evaluating a loop form containing an appropriate iteration-driving clause followed by a **collect** *variable* clause.

Caveat on operator usage. Applying an operator to "bad" operands (operands that do not satisfy the assumptions made by the operator about its operands) may lead to arbitrarily serious, and possibly mysterious, failures.

[15] An object being "passed" as an operand to an XLMS operator is represented by a *pointer*, in accordance with LISP convention.

3.1 Operators Pertaining to Concepts

(conceptp *object***)**

determines whether or not *object* is a concept, and returns **t** or **nil** accordingly. In many implementations of XLMS, it is quite costly to determine that an object is a concept when it is not even known to be a node.

(concept-nodep *node***)**

is equivalent to (**conceptp** *node*), but is faster because it can assume that its operand is a node. In fact, in some implementations of XLMS, **concept-nodep** may be an order of magnitude or more faster, on the average, than **conceptp**.

(make-concept *ilk tie cue***)**

creates a concept with the specified *ilk*, *tie*, and *cue*, if such a concept does not already exist in memory. In any case, the specified concept is returned. Note that creation of a concept could involve nothing more than turning a non-concept node into a concept node.

Recall that a particular primitive tie, which in a parenthesis-delimited LMS notational expression could be specified by just *letter*, would, as an argument, be specified by [**letter*].

(ilk *concept***)**, **(tie** *concept***)**, and **(cue** *concept***)**

return the ilk, tie, and cue of *concept*, respectively. These operations are trivial to perform in typical implementations of XLMS.

(same-ilk-and-tie *concept-A concept-B***)**

determines whether or not *concept-A* and *concept-B* have both the same ilk and the same tie, and returns **t** or **nil** accordingly. It is equivalent to (**and** (**eq** (**ilk** *concept-A*) (**ilk** *concept-B*)) (**eq** (**tie** *concept-A*) (**tie** *concept-B*))), assuming *concept-A* and *concept-B* are free of side effects, but is substantially faster in many implementations of LMS.

(primitive-tiep *tie***)**

determines whether or not *tie* is a primitive tie, and returns **t** or **nil** accordingly. Operand *tie* must be a concept.

(define-primitive-tie *primitive-tie-letter descriptive-atomic-symbol*)

defines the primitive tie which is designated by *primitive-tie-letter* and described (and distinguished from other primitive ties) by *descriptive-atomic-symbol*. Neither argument is interpreted; both must be atomic symbols. If one of the arguments has been used previously to designate or describe a primitive tie, then the other must also have been used to designate or describe that same primitive tie.

A primitive tie must be defined before LMS notation in which its designating letter appears can be absorbed. For descriptions of the primitive ties that are predefined in XLMS, see the appendix entitled "Predefined Primitive Ties and Attachment Relations".

3.1.1 The $\subset$ Relation and Operators that Use It

All but one of the operators described in this section apply to nodes in general, rather than just to concepts. They are listed here because they are applied most frequently to concepts, especially by experimenting users.

For two arbitrary nodes *a* and *b*, $a \subset b$ iff one of the following conditions hold:

(1) (**genus** *a*) = *b*, where *a* ≠ **summum-genus**;

(2) (**genus** *a*) $\subset$ *b*; or

(3) (**genus** *a*) = (**genus** *b*) and (**specializer** *a*) $\subset$ (**specializer** *b*), where (**specializer** *a*) and (**specializer** *b*) are both nodes.

Where *a* and *b* are both concepts, these conditions may be restated[16] as follows:

(1) (**ilk** *a*) = *b*, where *a* ≠ **summum-genus**;

(2) (**ilk** *a*) $\subset$ *b*; or

(3) (**ilk** *a*) = (**ilk** *b*), (**tie** *a*) = (**tie** *b*), and (**cue** *a*) $\subset$ (**cue** *b*), where (**cue** *a*) and (**cue** *b*) are both nodes.

[16] This definition of $\subset$ for concepts is valid only if the following constraint is observed: given a concept (*ilk***tie cue*), if *x* is a node such that (*ilk tie*) $\subsetneq x \subset$ *ilk*, then *x* must *not* be a concept. It is quite unlikely that this constraint would be accidentally violated.

The $\subset$ relation has the following properties worthy of note:

$a \subseteq$ **summum-genus**, for any node a;

$(a\ (b\ c)) \subset (a\ b) \subset a$;

$(i_1{*}t_1\ (i_2{*}t_2\ c)) \subset (i_1{*}t_1\ i_2) \subset i_1$, e.g., **(color*r (block*l 1))** $\subset$ **(color*r block)** $\subset$ **color**;

by condition (3), the condition for *derivative subclassification*, a set of nodes (concepts), all of which have the same genus (ilk and tie), is isomorphic under $\subset$ to the set of its specializers (cues);

whether $a \subset b$ holds or not is "memory-independent"; and

(*ilk*tie cue*) $\subset$ (*ilk tie*), if (*ilk*tie cue*) is implemented either as ((*ilk tie*) *cue*) or as (*ilk* (*tie cue*)).

When $a \subset b$, a is said to be an *inferior* of b (*under b*), and b is said to be a *superior* of a (*above a*). Many, perhaps most nodes have no inferiors, but **summum-genus** is the only node that has no superiors. The superiors of a node are well-ordered by $\subset$. Thus, every node but **summum-genus** has an *immediate superior*,[17] the "least" of its superiors *which exist in memory*. If y is the immediate superior of x, then x is an *immediate inferior* of y. Note that the immediate inferiors of a node, and often also its immediate superior, may change as nodes are added to and removed from memory.

The set of all node/immediate-superior relationships (or, equivalently, the set of all node/immediate-inferior relationships) determines the current *node tree*, a tree of all nodes in memory, with **summum-genus** as the root. The node tree is used by XLMS for accessing concepts via their ilks (or nodes via their genuses) and as the basis for the canonical ordering of nodes. It is typically also used by LMS-based systems as a taxonomy of descriptions (a particular kind of abstraction hierarchy) and hence as a key determinant of inheritance (of attributes and meta-attributes).

(underp *node-A node-B***)**

tests whether *node-A* $\subset$ *node-B*, and returns **t** or **nil** accordingly. (In earlier versions of LMS, **underp** was called **subclassp** and sometimes **subconceptp**.)

[17] In earlier versions of LMS, the immediate superior of a node was referred to as its "generalizer".

(underp-or-eq *node-A node-B***)**

tests whether *node-A* $\subseteq$ *node-B*, and returns **t** or **nil** accordingly. For two arbitrary nodes *a* and *b*, $a \subseteq b$ iff either $a \subset b$ or (**eq** *a b*). (In earlier versions of LMS, **underp-or-eq** was called **classp**.)

(... **for** *variable* **being superiors of** *node* ...),

(... **for** *variable* **being superiors of** *node* **under** *superior* ...),

(... **for** *variable* **being superiors of** *node* **under-or-eq** *superior* ...),

(... **for** *variable* **being** *node* **and its superiors** ...),

(... **for** *variable* **being** *node* **and its superiors under** *superior* ...), and

(... **for** *variable* **being** *node* **and its superiors under-or-eq** *superior* ...)

produces the superiors of *node* ($\subset$ or $\subseteq$ *superior*, where so specified) which exist in memory, if any, in $\subset$ order, with *node* also produced as the first value when an **and its superiors** variant is used. Operand *superior*, if present, must be a superior of *node*, except in the case of the last variant above, where it must be a superior of or **eq** to *node*. Note that the superiors of a node will typically include *non*-concept as well as concept nodes, and are *not* independent of what nodes exist in memory. The superiors of a node are trivial to produce in typical implementations of XLMS.

(... **for** *variable* **being immediate-inferiors of** *node* ...)

produces the immediate inferiors of *node*, if any, in canonical order. Note that the immediate inferiors of a node may change as nodes are added to or removed from memory. Note also that the immediate inferiors of a concept are not necessarily, or even frequently, concepts. The immediate inferiors of a node are easy to produce in typical implementations of XLMS.

By using the **immediate-inferiors for** form recursively, it is possible to visit all concepts in memory inferior to a given node. Observe that a **loop** form of the form

(loop for *subnode* **being immediate-inferiors of** *node*
    do <process *subnode*> <recur with *subnode* as *node*>)

will traverse, in "preorder" but excluding *node*, the node tree subtree whose root is *node*, and thus will visit every inferior of *node*.[18] (In fact, this traversal will visit the inferiors of *node* in canonical order!) Therefore, if <process *subnode*> is replaced by (**cond** ((**concept-nodep** *subnode*) <process *subnode*>)), precisely those concepts which are inferior to *node* will be processed.

(**look-for-least-superior-concept** *concept*)

returns the least concept x in memory, if any, such that *concept* $\subset x$. Unless *concept* is **summum-genus**, there is such a least concept because (a) **summum-genus** is such a concept, and (b) *any* subset of the superiors of *any* node is well-ordered by $\subset$. If *concept* is **summum-genus, nil** is returned. Note that the least superior concept of a concept may change as concepts are added to or removed from memory.

(**least-common-superior-concept** *concept-A concept-B*)

returns the least concept x in memory such that *concept-A* $\subseteq x$ and *concept-B* $\subseteq x$. There is such a least concept because (a) **summum-genus** is always such a concept, and (b) *any* subset of the superiors of *any* node is well-ordered by $\subset$. Note that the least common superior concept for two concepts may change as concepts are added to or removed from memory. Note also that "least common superior concept" is somewhat of a misnomer, since *concept-B* will be returned if *concept-A* $\subseteq$ *concept-B*, and *concept-A* will be returned if *concept-B* $\subseteq$ *concept-A*.

3.1.2 Operators that Look For Concepts

A concept that exists in memory can be reliably *accessed* via its ilk and possibly via its cue, but never via its tie. A concept will *not* be reliably accessible via its cue only if its cue is a node which has ever had **common-cue** attached to it as a metacharacterization. Thus, for example, the concept (**block*f big**) may not be accessible via **big** if [**big #m common-cue**] has ever been absorbed. Accessing a concept via its cue, when possible, is typically faster than accessing it via its ilk. (In high-performance implementations of LMS, the average time for either of these "access methods" can be expected to depend logarithmically on how many concepts in memory share the particular ilk or cue in question.)

[18] See Knuth [5], sections 2.3.1 and 2.3.2, for an extended discussion of tree traversal.

In XLMS, access via the ilk involves following a chain of node-to-immediate-inferior links. Thus, the overall efficiency of this access path depends on how many immediate inferiors a node has, on the average. A user can keep this number low by avoiding situations in memory where a particular ilk/tie combination has been paired with a large number of distinct atoms used as cues (or where a particular node used as a genus has been paired with a large number of distinct atoms used as specializers). Some situations of this type are avoided by using more specialized ilks (or genuses), and some by using sequences in lieu of atomic cues (or specializers).

(look-for-concept *ilk tie cue***)**

looks for a concept in memory having the specified *ilk*, *tie*, and *cue*. If such a concept exists, it is returned; otherwise, **nil** is returned. Also, the special variable ***least-existing-superior** will be set to the least existing superior of the concept sought (necessarily $\subseteq$ *ilk*), unless **summum-genus** itself is being sought, in which case ***least-existing-superior** is set to **nil**. Note that **nil** is returned even when there exists a non-concept node in memory which would metamorphose into the concept sought if (**make-concept** *ilk tie cue*) were applied.

(look-for-concept-via-cue *ilk tie cue***)**

looks, via *cue*, for a concept in memory having the specified *ilk*, *tie*, and *cue*. If such a concept exists and is accessible via its cue, it is returned; otherwise, **nil** is returned. Normally, **look-for-concept-via-cue** is used only when it is known that if the concept sought exists in memory, it will be reliably accessible via its cue. Note that **nil** is returned even when there exists a non-concept node in memory which would turn into the concept sought if (**make-concept** *ilk tie cue*) were evaluated. Operand *cue* may be a node or an atomic symbol.

(... **for** *variable* **being concepts-with-cue of** *cue* ...) or

(... **for** *variable* **being concepts-with-cue of** *cue* **under** *superior* ...)

produces, in canonical order, the concepts, if any, that (a) have *cue* for a cue, (b) are accessible via their cues, and (c) are $\subset$ *superior* (where so specified). Such an iteration-driving clause is typically used only when it is known that any existing concept having *cue* for a cue is accessible via its cue. Operand *cue* may be a node or an atomic symbol.

3.2 Operators Pertaining to Attachments

Whenever there is at least one attachment on a node *x* involving a particular attachment relation *r*, there exists a *zone* for *r* on *x*, which contains *all* such attachments involving *r*. A zone may be thought of either as a collection of attachments or as a collection of attached objects. Note that attachments can occur on all nodes, not just on concepts, and that attachments on nodes are not necessarily permanent.

The attachments (attached objects) constituting a zone are always well-ordered according to some attachment-relation-dependent rule. In LMS, the most common rule (though it can apply only to zones that can contain only nodes) is that the attached objects are always in canonical order. Where this rule does not apply, attachments are typically kept either in order of attachment or in *reverse* order of attachment.

In most zones, a particular object may be the attached object in at most one attachment, but in an **#sequence-element**[19] zone, for example, a given object may participate in any number of attachments.

An attachment relation argument may be specified as an attachment relation name in square brackets. Thus, for example, one might write **(make-attachment [#c] [scientist] person)**.

(make-attachment *attachment-relation object node*)

attaches *object* to *node* per *attachment-relation*, and returns *object*, provided either (a) such an attachment is not already present or (b) more than one such attachment is allowed for *attachment-relation*. Otherwise, nothing is done, and **nil** is returned.

Attachment of a new first object in a zone is accomplished by replacing the first object, after first inserting a copy of it after it--thus preserving the location of the zone at the expense of a somewhat peculiar behavioral interaction with zone element iteration.

(detach-if-present *attachment-relation object node*)

eliminates an attachment of *object* to *node* per *attachment-relation*, and returns *object*, if such an attachment exists. If there is more than one such attachment, the first in the zone is eliminated. If there is *no* such attachment, nothing is done and **nil** is returned.

[19] **#sequence-element** is a name for an attachment relation used privately by LMS.

Detachment of the first object in a zone, when there is more than one, is accomplished by replacing the first by a copy of the second, then deleting the second--thus preserving the location of the zone at the expense of a somewhat peculiar behavioral interaction with zone element iteration.

3.2.1 Operators that Get or Look For Attachments

(**get-nth-attachment** *n attachment-relation node*)

returns the *n*th attached object in the zone for *attachment-relation* on *node*. Operand *n* must be an *atomic small integer* (a "fixnum" in MACLISP), and *node* must have a zone for *attachment-relation* containing at least *n* attachments.

(**look-for-attachment** *attachment-relation object node*)

looks to see whether *object* is attached per *attachment-relation* to *node*. If so, *object* is returned; if not, **nil** is returned. Operand *object* may be any object that could validly be attached per *attachment-relation* to any node. Note that **look-for-attachment** can succeed only if *object* itself is found.

(**look-for-attachment-under-or-eq**
attachment-relation node-to-be-under-or-eq node)

looks for any node $\subseteq$ *node-to-be-under-or-eq* that is attached per *attachment-relation* to *node*. If there is just one such node, it is returned; if there are more than one, the first in the zone is returned; if there are none, **nil** is returned. This operator may be used only if zones for *attachment-relation* contain only nodes.

(**look-for-origin** *node*)

returns an object representing, and possibly describing, the *origin* of *node*, if known.[20] If the origin of *node* is unknown (that is, not recorded in memory), **nil** is returned.

[20] In the current MACLISP implementation of XLMS, the origin of a node is known if and only if it is a concept or sequence.

(... **for** *variable* **being each** *r* **of** *node* ...),

(... **for** *variable* **being each** *r* **of** *node* **under** *superior* ...),

(... **for** *variable* **being each** *r* **of** *node* **under-or-eq** *superior* ...),

(... **for** *variable* **being** *node* **and each** *r* ...),

(... **for** *variable* **being** *node* **and each** *r* **under** *superior* ...), or

(... **for** *variable* **being** *node* **and each** *r* **under-or-eq** *superior* ...)

produces the objects ($\subset$ or $\subseteq$ *superior*, where so specified) attached per attachment relation *r* to *node*, if any, in order of their appearance within the zone for *r*, with *node* also produced as the first value when an **and each** variant is used. A particular object might be produced more than once if it is the attached object in more than one attachment. An **under** or **under-or-eq** subclause may be used only if zones for the attachment relation *r* contain only nodes.

(**zone-present** *attachment-relation node*)

determines whether or not there is a zone present for *attachment-relation* on *node* (or, equivalently, whether or not there is at least one object attached per *attachment-relation* to *node*), and returns **t** or **nil** accordingly. This test, which is very fast in high-performance implementations of LMS, may be used to avoid unnecessary executions of **loop** forms of the form (**loop for** *variable* **being each** *attachment-relation* **of** *node* ...).

3.2.2 Operators Pertaining to Value Zones

(**get-value** *node-with-value-zone*)

gets the first (and typically only) attached object in the **#value** zone of *node-with-value-zone*. It is equivalent to (**get-nth-attachment 1 [#value]** *node-with-value-zone*), but is faster in typical implementations of XLMS, where the **#value** zone of a node is trivially accessible.

(**replace-value** *node-with-value-zone new-value*)

replaces the first (and typically only) attached object in the **#value** zone of *node-with-value-zone* by *new-value*. This operation is trivial to perform in typical implementations of XLMS, where the **#value** zone of a node is trivially accessible.

3.2.3 Operators Pertaining to Attachment Relations

Attachment relations have numbers as well as names: each attachment relation is identified by its distinct *attachment relation number*, a non-negative integer. When nodes are implemented in such a way that zones must appear in some order or other (as is the case in existing and planned implementations of LMS), the zones are kept in order of increasing attachment relation number. Zones with low attachment relation numbers will typically be faster to access than zones with high numbers. Some attachment relation numbers are reserved for relations declared by LMS itself, specifically, 0-11, 15-23, 31, and 35.[21]

(attachment-relationp *object*)

determines whether or not *object* is an attachment relation, and returns **t** or **nil** accordingly.

(declare-attachment-relation
attachment-relation-number attachment-relation-full-name
attachment-relation-single-letter-name-or-nil zone-features)

defines the attachment relation identified by *attachment-relation-number*. One or two names for the relation are specified, as atomic symbols, by arguments *attachment-relation-full-name* and *attachment-relation-single-letter-name-or-nil*, the latter being **nil** to indicate the absence of a single-letter name. Except for a prefix #, an attachment relation name must be normally spelled. The *attachment-relation-full-name* argument must have three or more characters in addition to the prefix #.

The *zone-features* argument is either **nil** or a list of atomic symbols that act as *zone feature indicators*. Possible zone feature indicators in LMS are: **concepts-only, nodes-only, canonical-order** (which implies **nodes-only**), **duplicates-allowed** (meaning that more than one attachment in a zone of this type may have the same attached object),

[21] In high-performance implementations of LMS, where each node has a special substructure to indicate what zones are present on it, operating efficiency may be considerably degraded when an attachment relation has been declared with a number greater than or equal to the number of digit bits in the simplest kind of atomic number (36 in the case of PDP-10 MACLISP, and 24 in the case of the M.I.T. LISP Machine). Thus, a user of LMS who is, or might ultimately be, concerned with efficiency should declare few attachment relations. Also, LMS users concerned with space efficiency should be aware that the amount of space required for nodes is minimized when the **#origin** attachment relation (used privately by LMS) is the highest-numbered attachment relation.

integral (meaning that a zone of this type is to be treated as an integral whole), and **a-list** (meaning that only "list cells" can be attached using this attachment relation, and that no two such list cells in a given zone of this type may have the same **car**).

Note that arguments in a **declare-attachment-relation** application are *not* interpreted. Thus, for instance, one might write

```
(declare-attachment-relation
     10 /#possible-cause /#z (concepts-only canonical-order)).
```

4. Specialized LMS Operators

The operators described in this chapter are more specialized than those described in the preceding chapter.

4.1 Operators Pertaining to Sequences and Seq. Fragments

Sequences and sequence fragments are used primarily as cues of concepts and as elements of larger sequences and sequence fragments, though they are also usable as independent objects and as specializers of nodes that are not concepts, sequences, or sequence fragments. However, a sequence or sequence fragment must *never* (except privately by LMS) be used as the ilk or tie of a concept, or as the genus of a node.

No sequence is ever an inferior of another sequence. But a sequence can be an inferior of a sequence fragment, and one sequence fragment can be an inferior of another. Specifically, a sequence $a_1, a_2, ..., a_n$ or a sequence fragment $a_1, a_2, ..., a_n, ...$ is an inferior of, or **eq** to, a sequence fragment $b_1, b_2, ..., b_m, ...$ if, and only if,

(1) $m \leq n$;

(2) $a_i = b_i$ for $i = 1, 2, ..., m\text{-}1$; and

(3) either $a_m \subset b_m$ or $a_m = b_m$.

Thus, for example, **[a, (b*p (c*t d)), "e"]** ⊂ **[a, (b*p (c*t d)), "e", ...]** ⊂ **[a, (b*p (c*t d)), ...]** ⊂ **[a, (b*p c), ...]** ⊂ **[a, b, ...]** ⊂ **[a, ...]** ⊂ **[...]**. Note that every sequence is an inferior of **[...]**.

A sequence that exists in memory can be reliably *accessed* always via a list of its elements, and sometimes via certain individual ones of its elements. A sequence will *not* be reliably accessible via a particular one of its elements only if that element (a) is an atomic symbol or (b) is a node which at some time has had **common-element** attached to it as a metacharacterization. Thus, for instance, the sequence **["foo", "bar"]** is never accessible via either element alone, and **[henry, the, eighth]** may not be accessible via **the** if **[the #m common-element]** has ever been absorbed. Accessing a sequence via an individual element, when possible, is typically faster than accessing it via a list of its elements. (In high-performance implementations of LMS, the average times for either of these access methods can be expected to depend logarithmically on how many sequences in memory share the individual element or the particular initial series of elements.)

Sequences have a distinctive dynamic property: the specialization that names a sequence may change as other sequences or sequence fragments are added to or removed from memory. Therefore, the genus and specializer of a sequence can change over time! This instability stems from XLMS's memory-saving practice of choosing naming specializations for sequences that are no more specific than is necessary to make every sequence have no inferiors in memory. As memory changes, the specialization that names a sequence may need immediate replacement, because it has just become insufficiently specific, or, conversely, it may become eligible for replacement, because it has become overly specific.

Sequence fragments exist in LMS primarily to provide a taxonomic structure above sequences, and secondarily to provide a completely reliable and rather uniformly efficient access method for sequences that does not involve access *via* elements.

(**LMS-sequencep** *object*)

determines whether or not *object* is a sequence, and returns **t** or **nil** accordingly. (This operator would have been called "**sequencep**" except for the fact that a primitive operator of that name exists in certain implementations of LISP that LMS might someday run under.)

(**sequence-fragmentp** *object*)

determines whether or not *object* is a sequence fragment, and returns **t** or **nil** accordingly.

(**sequence-nodep** *node*)

is equivalent to (**LMS-sequencep** *node*), but is faster because it can assume that its operand is a node. In fact, in some implementations of XLMS, **sequence-nodep** may be an order of magnitude or more faster, on the average, than **LMS-sequencep**.

(**sequence-fragment-nodep** *node*)

is equivalent to (**sequence-fragmentp** *node*), but is faster because it can assume that its operand is a node.

(**sequence-or-fragment-nodep** *node*)

is equivalent to (or (**sequence-nodep** *node*) (**sequence-fragment-nodep** *node*)), but may be significantly faster, because it is also equivalent to (**underp-or-eq** *node* [...])--a very fast operation in certain implementations of XLMS.

(make-sequence *list-of-elements***)**

converts *list-of-elements* into a sequence. This sequence may have existed previously. Note that only nodes and atomic symbols other than **nil** may appear in *list-of-elements*.

(LMS-append *series-of-elements*$_1$ *series-of-elements*$_2$ ...**)**

appends its operands to produce a series-of-elements result. (A *series-of-elements* operand or result is either a list, a sequence, a sequence fragment, or **nil**.) The type of the *last* operand determines the type of the result. If the last operand is a sequence or sequence fragment, the result will be a (possibly null) sequence or sequence fragment accordingly; otherwise, the result will be a list or **nil**. A last operand which is a list becomes part of the result, so that **LMS-append** will strictly subsume the standard LISP operator **append**. Thus, for instance, (LMS-**append** **[a, b]** **nil** **[c, ...]** '(**d** **e**)) would return a list ([**a**] [**b**] [**c**] **d** **e**), the two-element tail fragment of which would be identically the last operand.

Note that [], [...], or **nil** may be used as a last argument to force a particular type of result, without affecting the elements in the result. Hence LMS-**append** may be used to interconvert sequences, sequence fragments, and lists. For example, (LMS-**append** *sequence* [...]) would convert *sequence* into the corresponding sequence fragment. Also, (LMS-**append** *list-of-elements* []) is equivalent to (**make-sequence** *list-of-elements*), though the latter would undoubtedly be faster.

When LMS-**append** is used to add one or more elements to a sequence or sequence fragment, the result is necessarily a different object (though not necessarily a newly created one), since neither a sequence nor a sequence fragment can ever suffer a change of elements. Finally, note that when the result of an LMS-**append** operation is to be a sequence or sequence fragment, all elements of list operands must be either nodes or atomic symbols other than **nil**.

(get-nth-element-of-sequence *n sequence***)**

returns the *n*th element of *sequence*. Operand *n* must be an atomic small integer, and *sequence* must have at least *n* elements.

(get-last-element-of-sequence *non-null-sequence***)**

returns the last element of *non-null-sequence*.

(get-last-element-of-sequence-fragment *non-null-sequence-fragment***)**

returns the last element of *non-null-sequence-fragment.* In typical implementations of XLMS, this operation is trivial to perform.

(truncate-sequence-fragment *non-null-sequence-fragment***)**

returns the sequence fragment that results from removing the last element of *non-null-sequence-fragment.* For example, **(truncate-sequence-fragment [a, b, c, ...])** yields **[a, b, ...]**. In typical implementations of XLMS, a **truncate-sequence-fragment** operation is trivial to perform.

(look-for-sequence *list-of-elements***)**

looks for a sequence in memory having the same elements (in the same order) as *list-of-elements.* If such a sequence exists, it is returned; otherwise, **nil** is returned. Also, the special variable ***least-existing-superior** will be set to the least existing superior of the sequence sought (necessarily $\subseteq$ [...]). Note that only nodes and atomic symbols other than **nil** may appear in *list-of-elements.*

(look-for-sequence-via-element *element list-of-elements***)**

looks, via *element,* which must be a member of *list-of-elements,* for a sequence in memory having the same elements (in the same order) as *list-of-elements.* If such a sequence exists and is accessible via *element,* it is returned; otherwise, **nil** is returned. Normally, **look-for-sequence-via-element** is used only when it is known that if the sequence sought exists in memory, it will be accessible via *element.* Note that only nodes and atomic symbols other than **nil** may appear in *list-of-elements.*

(look-for-sequence-fragment *list-of-elements***)**

looks for a sequence fragment in memory having the same elements (in the same order) as *list-of-elements.* If such a sequence fragment exists, it is returned; otherwise, **nil** is returned. Also, except when [...] is returned, the special variable ***least-existing-superior** will be set to the least existing superior of the sequence fragment sought (necessarily $\subseteq$ [...]). Note that only nodes and atomic symbols other than **nil** may appear in *list-of-elements.*

(... **for** *variable* **being elements of** *sequence* ...)

produces the elements of *sequence*, if any, in order.

(... **for** *variable* **being sequences-with-element of** *element* ...) or

(... **for** *variable* **being sequences-with-element of** *element* **under** *sf* ...)

produces, in canonical order, the sequences, if any, that (a) have *element* as an element, (b) are accessible via *element*, and (c) are $\subset$ *sf* (where so specified). Such an iteration-driving clause is typically used only when it is known that any existing sequence having *element* for an element is accessible via *element*. Note that *element* must be either a node or an atomic symbol, that *sf* must be a node, and that no sequences will be produced if a *non*-sequence-fragment *sf* is specified.

(**sequence-length** *sequence*)

returns the length of *sequence*, as an atomic small integer.

(**sequence-fragment-length** *sequence-fragment*)

returns the length of *sequence-fragment*, as an atomic small integer. Note that (**sequence-fragment-length** [...]) is 0.

4.2 Low-Level Operators Pertaining to Nodes

The operators to be described in this section are needed only for working with *incidental nodes*: nodes that are not concepts, sequences, or sequence fragments. Only a few of the constructs described in this and the preceding chapter can produce incidental nodes.

(1) Applications of operators described in this section can produce incidental nodes.

(2) A **being superiors** or **being immediate-inferiors** iteration-driving clause can (and usually will) produce incidental nodes, though such nodes would typically be used only as "stepping stones" to *non*-incidental nodes.

(3) A **being each** iteration-driving clause could produce incidental nodes, but only insofar as they have been introduced and attached earlier.

(4) A **being elements** iteration-driving clause could produce incidental nodes, but only insofar as they have been introduced and used as sequence elements earlier.

Therefore, a user who avoids introducing incidental nodes (through notation, by atypical use of **being superiors** or **being immediate-inferiors** iteration-driving clauses, or by use of operators other than those described in this and the preceding chapter) may well have no need for the operators presented in this section.

Any node that exists in memory can be reliably *accessed* via its genus or via any *isogeneric superior* (a superior of the node that is an inferior of its genus); however, a node *cannot* be reliably accessed via its specializer *per se*. (If the node is a concept or sequence, there are also other methods for accessing it; see the sections entitled "Operators Pertaining to Concepts" and "Operators Pertaining to Sequences and Sequence Fragments"). In XLMS, access via the genus or an isogeneric superior involves following a chain of node-to-immediate-inferior links, using **immediate-inferiors** iteration paths. Thus, the overall efficiency of this access path depends on how many immediate inferiors a node has, on the average. See the subsection entitled "Operators that Look for Concepts" for a discussion of how a user can keep this number low.

(**nodep** *object*)

determines whether or not *object* is a node, and returns **t** or **nil** accordingly.

(**make-node** *genus specializer*)

creates a node with the specified *genus* and *specializer*, if such a node does not already exist in memory. In any case, the specified node is returned.

Creating a node consists of first creating a structurally appropriate *protonode*, then adding it to the node tree (that is, making it accessible via *genus*), at which point it ceases to be a protonode. When the public special variable ***create-common-superiors-spontaneously?** is non-**nil**, creating a node can entail the *spontaneous creation* of additional nodes which, at creation time, have at least two immediate inferiors. Note that **make-node** will not, under any circumstances, create a concept, though it may return an already existing one.

(**make-node-given-isogeneric-superior** *isogeneric-superior specializer*)

is equivalent to (**make-node** (**genus** *isogeneric-superior*) *specializer*), but is faster because accomplishing the latter operation consists of accomplishing the former operation after first moving down the node tree from (**genus** *isogeneric-superior*) to *isogeneric-superior*. The operands must be such that *specializer* $\subset$ (**specializer** *isogeneric-superior*). The node made will be an inferior of *isogeneric-superior*. Note that this operator depends on the $\subset$ relation.

(**genus** *node*) and (**specializer** *node*)

return the genus and specializer of *node*, respectively. These operations are trivial to perform in typical implementations of XLMS. They can depend on the state of memory *only* when *node* is a sequence; see the section entitled "Operators Pertaining to Sequences and Sequence Fragments".

(**immediate-superior** *node*)

returns the immediate superior, if any, of *node*: the least node x in memory such that *node* $\subset x$. Unless *node* is **summum-genus**, there is such a least node, because *any* subset of the superiors of *any* node is well-ordered by $\subset$. If *node* is **summum-genus**, **nil** is returned. Note that the immediate superior of a node may change as nodes are added to or removed from memory. In typical implementations of XLMS, this operation is trivial to perform.

(**least-common-superior** *node-A node-B*)

returns the least node x in memory such that *node-A* $\subseteq x$ and *node-B* $\subseteq x$. There is such a least node because (a) **summum-genus** is always such a node, and (b) *any* subset of the superiors of *any* node are well-ordered by $\subset$. In graph-theoretic terms, the *least common superior* of two nodes is the root of the smallest subtree of the node tree that includes them both. Note that the least common superior of two nodes may change as nodes are added to or removed from memory. Note also that "least common superior" is somewhat of a misnomer, since *node-B* will be returned if *node-A* $\subseteq$ *node-B*, and *node-A* will be returned if *node-B* $\subseteq$ *node-A*.

(**look-for-node** *genus specializer*)

looks for a node in memory having the specified *genus* and *specializer*. If such a node exists, it is returned; otherwise, **nil** is returned. Also, the special variable ***least-existing-superior** will be set to the least existing superior of the node sought (either *genus* or an isogeneric superior of the node sought), unless **summum-genus** itself is being sought, in which case ***least-existing-superior** is set to **nil**.

(look-for-node-given-isogeneric-superior *isogeneric-superior specializer***)**

is equivalent to (**look-for-node** (**genus** *isogeneric-superior*) *specializer*), but is faster because accomplishing the latter operation consists of accomplishing the former operation after first moving down the node tree from (**genus** *isogeneric-superior*) to *isogeneric-superior*. The operands must be such that *specializer* $\subset$ (**specializer** *isogeneric-superior*). The node made will be an inferior of *isogeneric-superior*. Note that this operator depends on the $\subset$ relation.

4.3 The Canonical Ordering of Nodes

People (and computers) have an easier time recognizing, comparing, and "understanding" collections that are consistently organized or, better yet, predictably organized. In LMS, collections of nodes are usually kept in *canonical order* when there is no specific reason to keep them in some other order. Canonically ordered collections of nodes are always consistently organized, and are often predictably organized as well.

Throughout this section, < will be used to denote the canonical order relation on nodes. The use of < is appropriate because the canonical ordering of nodes is a total ordering: for any two distinct nodes *a* and *b*, either $a < b$ or $b < a$. *Which* of these possibilities obtains may depend upon the past history of the memory that contains *a* and *b* (see below), but it will in any case continue to obtain as long as both *a* and *b* remain in that memory.

For two arbitrary nodes *a* and *b*, $a < b$ iff one of the following conditions (somewhat parallel to those that define the $\subset$ relation) hold:

(1) $a =$ (**genus** *b*), where $b \neq$ **summum-genus**;

(2) $a <$ (**genus** *b*);

(3) (**genus** *a*) $\subset$ (**genus** *b*) and (**genus** *a*) $< b$; or

(4) (**genus** *a*) = (**genus** *b*) and either

(a) (**specializer** *a*) < (**specializer** *b*), where (**specializer** *a*) and (**specializer** *b*) are both nodes, or

(b) (**specializer** *a*) is an atomic symbol and (**specializer** *b*) is a node, or

(c) (**specializer** *a*) and (**specializer** *b*) are both atomic symbols and *a* was created before *b*.

Where a and b are both concepts, these conditions may be restated[22] as follows:

(1) a = (**ilk** b), where $b \neq$ **summum-genus**;

(2) a < (**ilk** b);

(3) (**ilk** a) ⊂ (**ilk** b) and (**ilk** a) < b;

(4) (**ilk** a) = (**ilk** b) and (**tie** a) < (**tie** b); or

(5) (**ilk** a) = (**ilk** b), (**tie** a) = (**tie** b), and either

(a) (**cue** a) < (**cue** b), where (**cue** a) and (**cue** b) are both nodes, or

(b) (**cue** a) is an atomic symbol and (**cue** b) is a node, or

(c) (**cue** a) and (**cue** b) are both atomic symbols and a was created before b.

Condition (5c) above says that sets of concepts having a common ilk and tie and distinct atomic symbol cues are canonically ordered according to their *age* rather than, say, an **alphalessp**[23] ordering of their cues. (Condition (4c) makes a similar statement about nodes in general.) This permits a concept taxonomist to impose a particular canonical ordering on such a set s, which in turn will determine "derivatively" (see property 2 below) the canonical ordering of any set of nodes that all have the same naming specialization except in some one position where different members of s appear. (XLMS itself takes advantage of this to get the canonical ordering of positive-integer-labeled concepts to correspond to numerical order.)

The canonical ordering of nodes in a node tree can be defined *intuitively* as the order in which they would be visited during a preorder traversal of the tree, where nodes having a common immediate superior s are visited in the following order: first those with atomic symbol specializers, in order of creation; then those with nodal specializers and genus s, ordered according to the canonical ordering of their specializers; and finally those with nodal specializers and a genus *other than s*, ordered

[22] This definition of < for concepts is valid only if the following constraint (also noted in the definition of ⊂ for concepts) is observed: if (*ilk***tie cue*) is a concept, and if x is a node such that (*ilk tie*) ⊆ x ⊂ *ilk*, then x must *not* be a concept. Furthermore, if (*ilk***tie cue*) were to be implemented as (*ilk* (*tie cue*)), condition (4) would have to be revised for the cases of (**tie** a) ⊂ (**tie** b) and (**tie** b) ⊂ (**tie** a).

[23] **alphalessp** is a primitive LISP predicate used for alphabetic and lexicographic ordering.

according to the canonical ordering of their specializers.

The canonical order relation on nodes has the following useful properties.

(1) If $a < b$ in a given state of the memory containing a and b, then it will remain the case that $a < b$ as long as both a and b remain continuously in that memory.

(2) For any two nodes x and y whose naming specializations differ only in that x has x_c where y has y_c, $x < y$ if $x_c < y_c$. For example, if **little** < **big**, then **(block*f little)** < **(block*f big)** and **((block*f little)*u scene-1)** < **((block*f big)*u scene-1)**.

(3) Canonically ordered collections of nodes are often predictably organized.

(4) In all canonically ordered collections of nodes, the nodes, if any, that are inferior to a given node will appear in the same "relative position". Thus, simple common-superior matching in such collections may be efficiently accomplished using binary search and parallel scanning techniques. This is especially important in serial-machine implementations of LMS, of which XLMS is an example.

(5) The canonical order relation is useful in defining canonical representations. For example, the canonical representation of an itemized set might be a concept, the cue of which is a sequence having the set elements in canonical order.

(6) A $<$ or $\subset$ test can be performed trivially *when both operands are located in CO-LTM* ("canonically-ordered long-term memory"), an area of memory wherein nodes are kept in canonical order with "subtree extent pointers". For $<$, an address comparison suffices. For $\subset$, an address range test suffices.[24] The address range test is based on the following simple theorems relating $\subset$ and $<$: (1) $b \subset a$ implies $a < b$; (2) $a < b < c$ and $c \subset a$ together imply $b \subset a$.

(beforep *node-A node-B*)

tests whether *node-A* < *node-B*, that is, whether *node-A* precedes *node-B* in the canonical ordering of nodes, returning **t** or **nil** accordingly. Note that **beforep**, as well as many operators that depend on **beforep**, can be expected to be relatively slow when most nodes in memory are *not* in LTM.

[24] See Knuth [5], section 2.3.3 and exercise 12 of section 2.4.

(beforep-or-eq *node-A node-B*)

tests whether *node-A* $\leq$ *node-B*, and returns **t** or **nil** accordingly. For two arbitrary nodes *a* and *b*, $a \leq b$ iff either $a < b$ or (**eq** *a b*).

4.4 Operators Pertaining to Atomic Symbols

Atomic symbols are used in XLMS to represent lexical units (words, phrases, affixes, etc.), characters, and labels. Atomic symbols that represent lexical units typically appear in memory as concept cues and sequence elements.

(make-character-code *character-or-character-name*)

converts *character-or-character-name* into a *character code* (an atomic small integer). The *character-or-character-name* operand is either a single-character atomic symbol or a *character name* (one of the atomic symbols **eof, bell, backspace, tab, linefeed, formfeed, newline,** or **space**). The character-code result must be interpreted with respect to a particular implementation-dependent *character code set.* Any such character code set must be such that (a) the codes for digits are consecutive and in numerical order, (b) the codes for lower-case letters are consecutive and in alphabetical order, and (c) the codes for upper-case letters are consecutive and in alphabetical order.

Through the use of **make-character-code**, character codes can be represented code-set-independently. For example, **[!(make-character-code 'tab)]**, in LISP or LMS notation, represents the code for a tab character, in any code set.

(make-character *character-code*)

converts *character-code* into a single-character atomic symbol. The *character-code* operand is interpreted with respect to a particular implementation-dependent character code set.

(make-LMS-atomic-symbol *atomic-symbol*)

converts *atomic-symbol* to the corresponding LMS atomic symbol. (This would be an identity operator if it did not map LISP's **nil** into LMS's "**nil**".)

(normal-spellingp *atomic-symbol***)**

determines whether or not *atomic-symbol* is normally spelled, and returns **t** or **nil** accordingly. In many implementations of LMS, this operation is very quick to perform when *atomic-symbol* is being used to represent a global label.

(make-global-label-assignment *global-label object***)**

assigns the global label represented by *global-label*, an atomic symbol, to *object*, superceding any prior assignment of the label. The *global-label* operand must have the same spelling as the global label it represents, and thus must be normally spelled.

(look-for-global-label *object known-to-be-a-node?***)**

looks for a global label assigned to *object*. If *object* is a node and such a global label exists, the atomic symbol having the same spelling is returned; otherwise, **nil** is returned. If *known-to-be-a-node?* is non-**nil**, *object* must be a node.

(list-global-labels *assignment-selection-predicate? alphabetize?***)**

returns a (possibly empty) list of atomic symbols representing global labels, arranged in alphabetical order if *alphabetize?* is absent or non-**nil**. The elements of this list represent, without duplication, *all* global labels assigned to objects in memory that pass the following selection test: a global label passes if *assignment-selection-predicate?* is absent or **nil** or if *assignment-selection-predicate?* applied to the assignment of the label returns a non-**nil** value. A non-**nil** *assignment-selection-predicate?* must be "funcallable" with one operand. Note that the second operand of **list-global-labels** is optional, as is the first when the second is not present.

4.5 Operators Pertaining to Numerically Labeled Concepts

Numerically labeled concepts are canonical representations of numbers as concepts. As might be expected, every numerically labeled concept has a numeric label (in fact, arbitrarily many numeric labels, since a numeric label may have arbitrarily many leading zeros). A numerically labeled concept represents the number expressed by its numeric labels.

No *particular* details of the specializations that name numerically labeled concepts will be provided here. However, it may be assumed that the canonical ordering of positive-integer-labeled concepts corresponds to numerical order.

To do arithmetic and comparison operations on numerically labeled concepts, convert them to atomic numbers first. For example, to determine whether two numerically labeled concepts *m* and *n* representing small integers differ by less than ten, do, in MACLISP,

(< (abs (- (make-atomic-number *m*) **(make-atomic-number** *n*))) **10).**[25]

Note that *atomic* numbers, though relatively economical[26], may not, for reasons of uniformity, be used as ilks, ties, cues, genuses, specializers, or attached objects in **concepts-only** or **nodes-only** zones. But note that atomic numbers *can* be attached to nodes as values.

Within an LMS-based system, one might find, in addition to atomic numbers and numerically labeled concepts, many other forms of representation for numbers; in fact, one might find many different forms of representation for numbers *as concepts.*

(numerically-labeledp *node*)

determines whether or not *node* is a numerically labeled concept, and returns **t** or **nil** accordingly. Note that the operand must be a node.

(numerically-spelledp *atomic-symbol*)

determines whether or not *atomic-symbol* is numerically spelled, and returns **t** or **nil** accordingly. For a precise specification of the syntax of numeric spellings, see the appendix entitled "BNF Summary of LMS Notation".

(make-atomic-number *numerically-labeled-concept*)

converts *numerically-labeled-concept* into an atomic number. This atomic number may have existed previously, but the mere existence in memory of a suitable atomic number to return does not ensure that it, rather than some newly created equivalent atomic number, will be returned. In many implementations of XLMS, this operation is trivial to perform.

[25] Note that numbers in LISP notation in the source text for LMS modules are interpreted in base ten by default.

[26] In XLMS, numerically labeled concepts are roughly an order-of-magnitude costlier to create and keep in memory than are atomic numbers.

(make-numerically-labeled-concept *atomic-number***)**

converts *atomic-number* into a numerically labeled concept. This operation can be costly to perform, especially when the concept to be returned does not already exist in memory.

Appendix I - BNF Summary of LMS Notation

In the variant of BNF used below, terminal symbols appear in boldface. Curly brackets enclose optional or repeatable items, where absence of a suffix means optional, an * suffix means optional or repeatable any number of times, and a + suffix means repeatable any number of times.

```
LMS-notation-as-LISP-expression ::=
    square-bracketed-expression | absorption-form

square-bracketed-expression ::=
    complex | [subject] | [trivial-itemization]

complex ::= [subject body-of-complex]

absorption-form ::=
    [% subject {body-of-complex}] | [% trivial-itemization]

subject ::= expression | *primitive-tie-letter | attachment-relation-name

body-of-complex ::= {{attachment-specification} expression}+

expression ::= unit-expression | equation | non-trivial-itemization

equation ::= side = side |
             global-label = (ilk-expression*tie-expression ") |
             global-label = (genus-expression ")

side ::= item | non-trivial-itemization

non-trivial-itemization ::= {item ,}+ item | {item ,}+ ...

trivial-itemization ::= item,

item ::= unit-expression | order-insensitive-itemization

order-insensitive-itemization ::= {unit-expression &}+ unit-expression

unit-expression ::= root-expression | double-quoted-spelling |
                    anaphoric-abbreviation | computational-expression

root-expression ::= label | delimited-expression

delimited-expression ::=
    square-bracketed-expression |
    (ilk-expression*tie-expression cue-expression) |
    (genus-expression specializer-expression)
```

```
ilk-expression ::= label | delimited-expression | anaphoric-abbreviation

tie-expression ::= primitive-tie-letter | root-expression

primitive-tie-letter ::= letter

cue-expression ::= expression | trivial-itemization

genus-expression ::= expression

specializer-expression ::= expression | trivial-itemization

anaphoric-abbreviation ::=
    {:}+ | root-expression{:}+{instance-specifier}

instance-specifier ::= root-expression | double-quoted-spelling | u | U

computational-expression ::= !LISP-form

attachment-specification ::=
    attachment-relation-name | reverse-attachment-specification |
    two-way-attachment-specification | #

attachment-relation-name ::= #attachment-relation-name-root

reverse-attachment-specification ::= attachment-relation-name-root#

two-way-attachment-specification ::=
    attachment-relation-name-root#attachment-relation-name-root |
    #attachment-relation-letter attachment-relation-letter

attachment-relation-name-root ::=
    attachment-relation-letter |
    normal-character normal-character {normal-character}+

attachment-relation-letter ::= letter

label ::= global-label | local-label

global-label ::= normal-spelling

local-label ::= $normal-spelling

double-quoted-spelling ::=
    "normal-spelling{"} | ""{character-in-spelling}*"

character-in-spelling ::= """" | any-character-but-double-quote

normal-spelling ::= {normal-character}+

normal-character ::= letter | digit | - | . | '
```

letter ::= lower-case-letter | upper-case-letter

lower-case-letter ::= a | b | ... | z

upper-case-letter ::= A | B | ... | Z

digit ::= 0 | 1 | ... | 9

spelling ::= label | double-quoted-spelling |
            attachment-specification | primitive-tie-letter

opening-delimiter ::= [| (

closing-delimiter ::=] |)

connective ::= = | , | &

numeric-label ::= numeric-spelling

numeric-spelling ::= integer-spelling | number-with-decimal-point |
            number-in-scientific-notation

integer-spelling ::= {-}{digit}+

number-with-decimal-point ::= {-}{digit}*.{digit}+

number-in-scientific-notation ::=
    number-with-decimal-point e integer |
    number-with-decimal-point E integer

As a general rule, at least one space-or-equivalent must appear between adjacent spellings, and any number of space-or-equivalents may optionally appear before or after any terminal character that is *not* part of a spelling. Exceptions to this rule are: (a) a space-or-equivalent may *not* appear after an *, within a [%, or between components of an anaphoric-abbreviation; (b) at least one space-or-equivalent must appear before a bare " or before a ! that is not immediately preceded by an opening-delimiter or a connective; (c) at least one space-or-equivalent must appear before a : that is not immediately preceded by an opening-delimiter, a connective, or the root-expression in an anaphoric-abbreviation; and (d) at least one space-or-equivalent must appear after a : that is not immediately followed by an *, a connective, a closing-delimiter, or an instance-specifier. Note that any number of space-or-equivalents may appear *before* an *, in conformance with the general rule.

The syntax given above is not quite complete and precise, in that: (a) LISP-form, space-or-equivalent, and any-character-but-double-quote are left undefined; (b) no rules are given for space-or-equivalents before or after LISP-forms; (c) a trivial-itemization can appear before the] of a complex or, in some contexts, as a side (of an equation); (d) an item terminating a non-trivial-itemization cannot be the label ...; and (e) a root-expression appearing as an instance-specifier cannot be the label u or U.

Appendix II - The MACLISP Implementation of XLMS

The MACLISP implementation of XLMS may be invoked in ITS[27] by typing either XLMS^K or :XLMS followed by a carriage-return. After XLMS has announced its version number and has printed an asterisk, the user may proceed to enter forms online as he or she would to the **read-eval-print** loop of LISP.

Square-bracket-delimited expressions representing concepts, sequences, sequence fragments, nodes, dummy nodes, or attachment relations may be typed online, because such objects "evaluate to themselves" in LISP. Such an object is *printed* online as a square-bracket-delimited expression that normally reveals the object's immediate constituents, its labels (if any), and selected objects associated with it (e.g., attachments on it). Thus, to "examine" such an object, a square-bracket-delimited expression representing it need only be typed online.

If a negative "fixnum", -*n*, is typed online, the *n*th previous object printed online will be reprinted. If a positive fixnum *n* is typed online, the most-recently-printed *n*th component of a square-bracket-delimited expression will itself be printed online.

In typing LMS notation, the user may sometimes find that the "rubout processing" provided by XLMS (perhaps inherited directly from MACLISP) does not appear to be functioning properly. When a problem of this kind is encountered, the safest procedure is to abort the reading process by typing a ^G, thereby restarting the **read-eval-print** loop.

The failure to define a label can cause unforeseen difficulties in code not designed to deal with dummy nodes. Therefore, the following two "online operators" are typically used as a matter of course during the development of a knowledge base.

(ugl)

returns a (possibly empty) list of atomic symbols representing all global labels that have been used but not yet defined. This list is arranged in alphabetical order and contains no duplicates.

[27] *ITS* is the operating system used on M.I.T.'s ML, AI, and MC machines, all of which are modified PDP-10s.

(defugl)

causes all global labels that have been used but not yet defined to be given (permanent) default definitions. (A *default definition* is a concept whose ilk is **default-definition**.) Immediately after this has been done, it is possible to determine quite straightforwardly whether any labels have previously been given "cyclical" definitions; **defugl** does this, and issues a warning if any cyclically defined labels are found.

XLMS-based systems are normally constructed using LSB. See the LSB manual [3] for how to: set up an XLMS-based system using LSB, format a source file (for a module), compile a module, load the system, etc.

References

1. Brown, G. P. "A Framework for Processing Dialogue", MIT/LCS/TR-184, MIT Laboratory for Computer Science, Cambridge, Ma., July 1977.

2. Burke, G. and Moon, D. "LOOP Iteration Macro", MIT/LCS/TM-169, MIT Laboratory for Computer Science, Cambridge, Ma., July 1980.

3. Burke, G. "LSB Manual" (in preparation), MIT Laboratory for Computer Science, Cambridge, Ma.

4. Hawkinson, L. "The Representation of Concepts in OWL", *Proceedings, Fourth International Joint Conference on Artificial Intelligence*, Tblisi, Georgia, USSR, Sept. 1975.

5. Knuth, D. E. *Fundamental Algorithms*, Vol. 1 of *The Art of Computer Programming*, second edition, Addison-Wesley, 1973.

6. Long, W. J. "A Program Writer", Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Ma., Nov. 1977 (available as MIT/LCS/TR-187, MIT Laboratory for Computer Science).

7. Martin, W. A. "The OWL Concept Hierarchy", in *Proceedings of Symposium of Directions in Artificial Intelligence: Natural Language Processing*, Grishman, R. (Ed.), Courant Computer Science Report #7, Courant Institute of Mathematical Sciences, Computer Science Department, New York University, New York, N. Y., Aug. 1975.

8. Martin, W. A. "OWL", *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, MIT, Cambridge, Ma., Aug. 1977, 985-87.

9. Martin, W. A. "Descriptions and the Specialization of Concepts", in *Artificial Intelligence, an MIT Perspective*, Winston, P. (Ed.), MIT Press, Cambridge, Ma., 1979.

10. Swartout, W. R. "A Digitalis Therapy Advisor with Explanations", MIT/LCS/TR-176, MIT Laboratory for Computer Science, Cambridge, Ma., Aug. 1977.

11. Szolovits, P., Hawkinson, L. B., and Martin, W. A. "An Overview of OWL, a Language for Knowledge Representation", in *Proceedings of Workshop on Natural Language for Interaction with Data Bases*, International Institute for Applied Systems Analysis (IIASA), Schloss Laxenburg, Austria, Jan. 1977 (also available as MIT/LCS/TM-86).

INDEX

*a 6
#a 9
a-list *25*
above *17*
absorb-time *12*
absorption form *13*
absorption (of LMS notation) *3*
abstraction hierarchy 17
access (to a concept) *19*
access (to a sequence) *26*
age (of a node) *34*
alphalessp *34*
anaphor *10*
anaphor (in LMS notation) *10*
anaphoric abbreviation 11
append 28
#applicable-function 9
appositive 6
assignment (of a label) *4*, 6, 37
atomic number 12, 38
atomic small integer *22*
atomic symbol 1, 3, 12, 36
attachment *2*, 7, 44
attachment relation *2*, 24
attachment relation name *8*, 21, 24
attachment relation number *24*
attachment-relationp *24*
attachment specification *8*
backspace *36*
beforep *35*
beforep-or-eq *36*
bell *36*
body (of a loop form) 14
#c 9
canonical-order *24*
canonical order (of nodes) *33*, 11, 19,
  21
character 36
character code *36*
character code set *36*
character name *36*
#characterization 9
CO-LTM *35*
colon anaphor *10*
common-cue 19
common-element 26
common-superior matching 35
complex *8*
computational expression *12*, 3
concept *1*, 5, 19
concept-nodep *15*
concept taxonomy 34
conceptp *15*
concepts-only *24*
concepts-with-cue *20*
constituent (of an object) *1*
cue *1*, 19
cue *15*
declare-attachment-relation *24*, 10
default definition *45*
default-definition *45*
define-primitive-tie *16*, 6
definition (of a label) *6*, 4
defugl *45*
delimited LMS notational expression
  5
derivative subclassification *17*
detach-if-present *21*
determinability (of what is represented
  by a piece of notation) *3*
ditto *7*
double-quoted spelling *3*
dummy node *2*, 7
duplicates-allowed *24*
#e 9
each *23*, 30
element (of a sequence) *1*
elements *30*
eof *36*
eq *6*
#equal 9
equation *6*
#exemplar 9
*f 6
#f 9
fixnum 22, 44
formfeed *36*
function 6
#function 9
generalizer 17
genus *1*, 31
genus *32*
get-last-element-of-sequence *28*

get-last-element-of-sequence-fragment *29*
get-nth-attachment *22*
get-nth-element-of-sequence *28*
get-value *23*
global label *4*, 37
#h 9
#has-role 9
*i 6
ilk *1*, 19
ilk *15*
immediate constituent *1*, 44
immediate inferior (of a node) *17*, 18
immediate-inferiors *18*, 30, 31
immediate-superior *32*
immediate superior (of a node) *17*
incidental node *30*, 2
individual 6
inferior (of a node) *17*
inheritance 17
integral *25*
#inverse-metacharacterization 9
isogeneric superior *31*
itemization *11*
iteration-driving clause *14*
ITS *44*
#k 9
#Katz-feature 9
knowledge base 44
knowledge-based system 2
knowledge representation 1
label *2*, 4, 36, 44
least common superior *32*
least-common-superior *32*
least-common-superior-concept *19*
***least-existing-superior** *20*, *32*
letter (in a normal spelling) 3
lexical unit 36
linefeed *36*
LISP 1
LISP notation 3
list cell 25
list-global-labels *37*
LMS *1*
LMS-append *28*
LMS notation 3
LMS Notation *3*
LMS notational expression *3*
LMS-sequencep *27*
local label *4*
look-for-attachment *22*
look-for-attachment-under-or-eq *22*
look-for-concept *20*
look-for-concept-via-cue *20*
look-for-global-label *37*
look-for-least-superior-concept *19*
look-for-node *32*
look-for-node-given-isogeneric-superior *33*
look-for-origin *22*
look-for-sequence *29*
look-for-sequence-fragment *29*
look-for-sequence-via-element *29*
loop 14
loop form 14
LSB 45
LTM 35
#m 9
MACLISP 44
make-atomic-number *38*
make-attachment *21*
make-character *36*
make-character-code *36*
make-concept *15*
make-global-label-assignment *37*
make-LMS-atomic-symbol *36*
make-node *31*
make-node-given-isogeneric-superior *31*
make-numerically-labeled-concept *39*
make-sequence *28*
memory *1*
#metacharacterization 9
multiset representation 11
#n 9
newline *36*
nil 4, 36
node *1*
node tree *17*, 18, 34
nodep *31*
nodes-only *24*
normal spelling *3*, 4, 37
normal-spellingp *37*
notation-based construction 13
null sequence 1, *11*
null-sequence *11*
null sequence fragment 1, 12

number 37
numeric label *4*, 37
numeric spelling *4*, 38
numerically labeled concept *37*, 4, 34
numerically-labeledp *38*
numerically-spelledp *38*
object *1*
online 44
order-insensitive itemization *11*
origin *22*
#origin 24
OWL 2
*p 6
#p 9
parenthesized pair *5*
partitive 6
path clause *14*
pointer *14*
#predicate-type 9
preorder traversal 18
primitive tie 6, 15
primitive-tiep *15*
printing online *44*
private *9*
proper name *1*
protonode *31*
prototype application *14*
#q 9
*r 6
#r 9
read-eval-print loop 44
reliable access (to a node) *31*
replace-value *23*
reverse-attachment specification *8*
role-in 6
#role-in 9
rubout processing 44
*s 6
#s 9
same-ilk-and-tie *15*
#semantic-name 9
semantic network 1
sequence *1*, 11, 27, 35
#sequence-element 21
sequence fragment *1*, 12, 27
sequence-fragment-length *30*
sequence-fragment-nodep *27*
sequence-fragmentp *27*
sequence-length *30*
sequence-nodep *27*
sequence-or-fragment-nodep *27*
sequences-with-element *30*
set representation 11
space *36*
specializer *1*, 31
specializer *32*
species 6
spontaneous creation (of a node) *31*
square-bracket-delimited expression *7*, 44
stereotype 6
subclassp 17
subconceptp 17
subtree extent pointer 35
summum-genus *1*, 17, 19, 32
superior 19, 32
superior (of a node) *17*, 18
superiors *18*, 30
#syntactic-name 9
*t 6
tab *36*
taxonomy of descriptions 17
tie *1*, 19
tie *15*
trivial itemization *11*
truncate-sequence-fragment *29*
two-way attachment specification *8*
*u 6
ugl *44*
under *17*
underp *17*
underp-or-eq *18*
unique-role-in 6
uparrow anaphor *10*
#v 9
#value 9
#x 9
XLMS *1*, 44
zone *21*
zone feature indicator *24*
zone-present *23*
≤ *36*
< *33*, 35
⊂ *16*, 17
⊆ *18*

OFFICIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center, Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washingon, D. C. 20374
1 copy